Viral Infections in Cisco IOS

Ariel Futoransky

Black Hat USA
Las Vegas, August 2008

Researchers

- Gerardo Richarte
  Corelabs
  Core Security Technologies

- Sebastián Muñiz
  Sr. Exploit Writer
  Core Security Technologies

- Ariel Futoransky
  Corelabs
  Core Security Technologies

Agenda

1. Introduction
2. The D.I.K. approach
3. Embedded analysis and other scenarios
4. Implications
5. Demo
6. Additional reflections

Introduction

Rootkits & IOS

Rootkits & Backdoors

- A **rootkit** is a program (or combination of several programs) designed to take fundamental control (in Unix terms "root" access, in Windows "Administrator" access) of a computer system, without authorization by the system's owners and legitimate managers

- A **backdoor** in a computer system (or cryptosystem or algorithm) is a method of bypassing normal authentication, securing remote access to a computer, obtaining access to plaintext, and so on, while attempting to remain undetected

Hardware & Security

- Hardware vs. Software
- Exactly where is the line between mutable and immutable?
- It is important to define the scenario and the threat model

Rootkits & Network Devices

- Considering that
  - There are known vulnerabilities affecting different network devices.
  - There are open projects that customize or completely reprogram network devices.
- And
  - Control of the network infrastructure could impact a lot of different layers.
  - A single router compromise could result in a complete subverting of the system.

Cisco IOS

- IOS is not an exception.
- Apparently there are no rootkits on the wild.
- It is of interest because of its widespread use in critical infrastructure components.
- Many of the ideas developed here could be applied to other network or embedded devices in general.

IOS Architecture

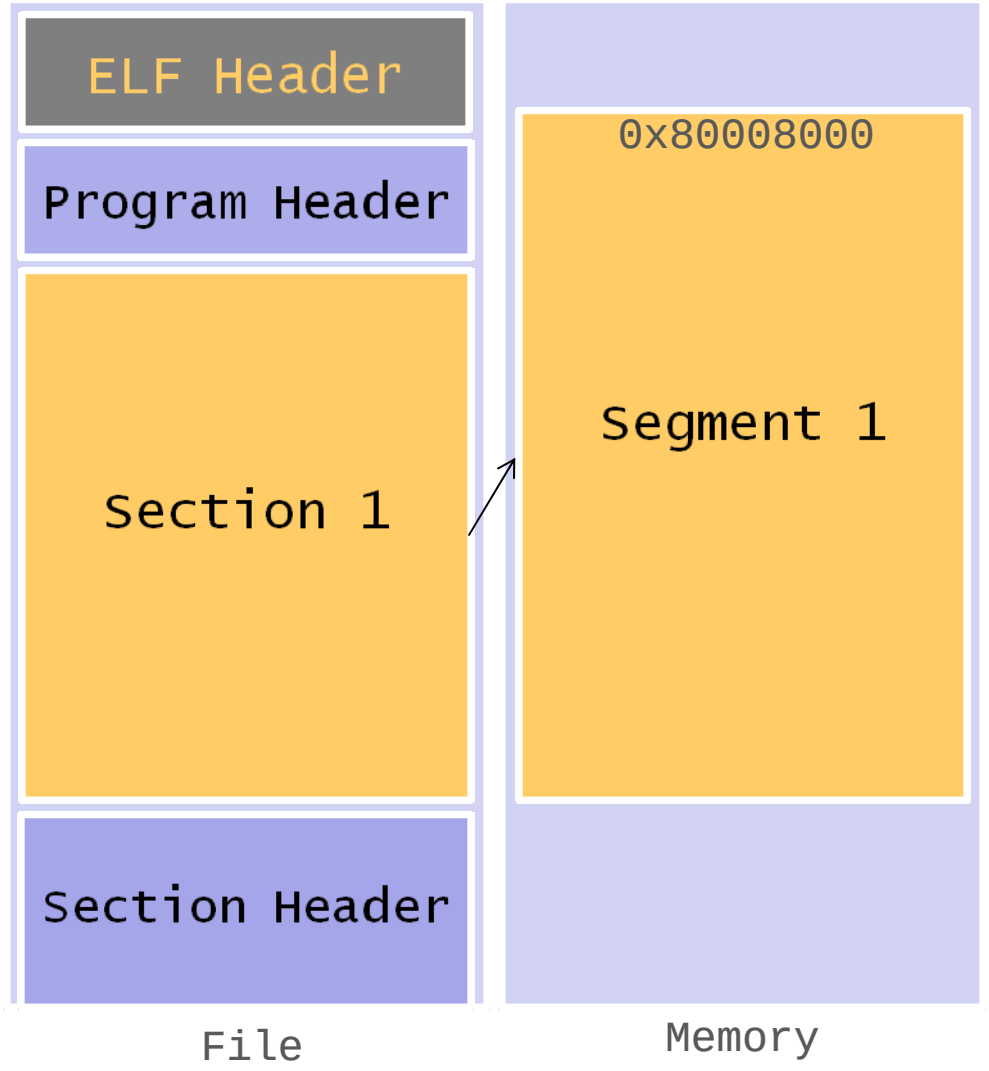
- Monolithic architecture which runs as a single image.
- All “processes” have access to each other’s memory.
- Uses 'run to completion' priority scheduling.
- FIFO (First In First Out) process queue.
- This model reduces local security and system stability.
- Completely different to modern OSes.

Creating a rootkit

- Locating API methods & data structures in memory (prototypes & addresses)
- Intercepting / Hooking
- Influence & manipulation
- Adding stealth functions

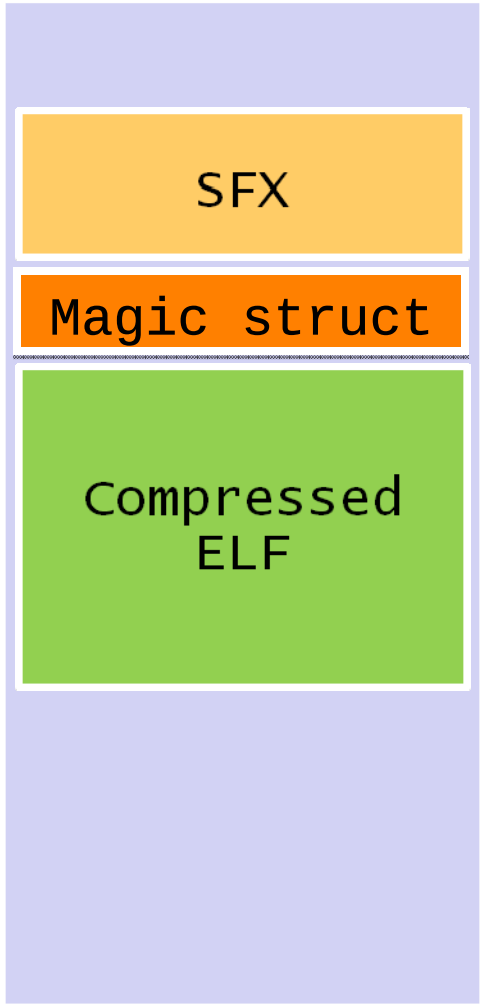
Binary Format

- ELF
- Extensive Linking Format
- Single file, single Image
- 1 Program Section
- Loaded at fixed Address



IOS Boot sequence

1. Boot loader performs POST and locates the ELF image in Flash.
2. The image is copied to RAM and the entry point is invoked
3. SFX code decompresses a larger image and transfers control to it.



DIK

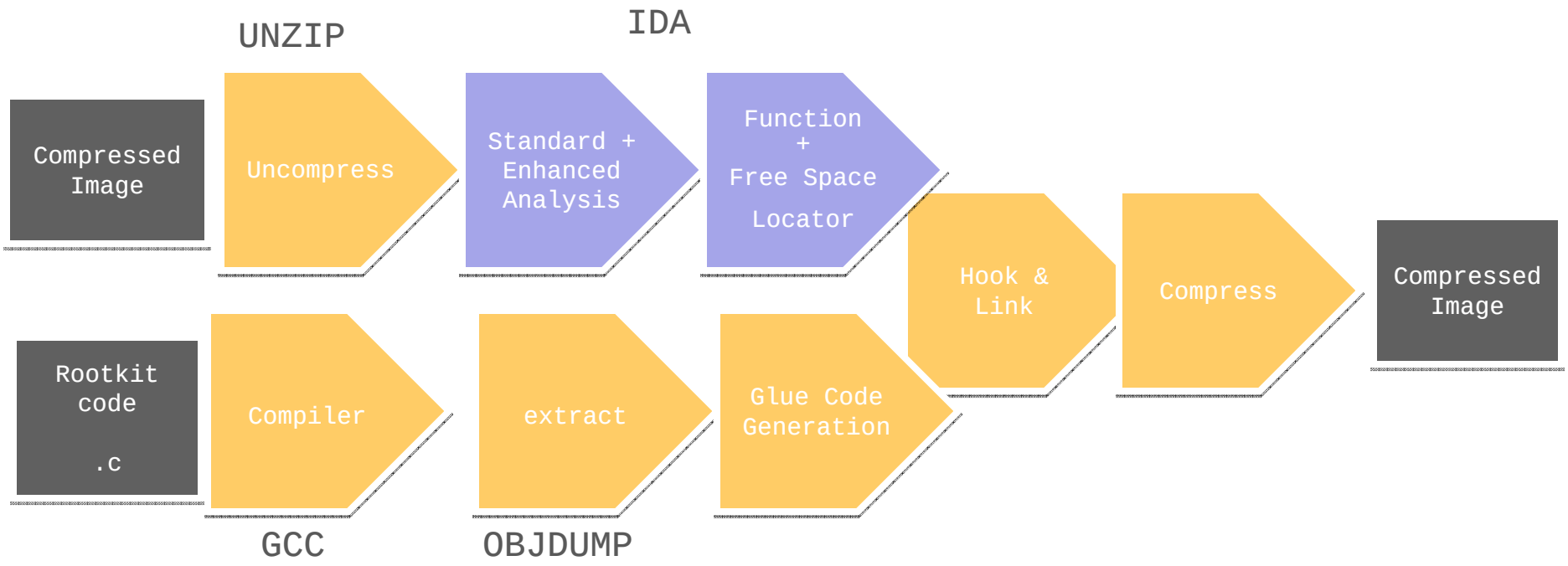
"Da IOS Rootkit"

Offline approach

The DIK Approach

- DIK automates the deployment of a rootkit code to different versions of IOS.
- Uses .c source for the specification of hooked function replacement
- Works for PPC & MIPS
- Depends heavily on IDA Pro
- Uses gcc/binutils

DIK Architecture



Get the Image

- Unzip can be used to extract the uncompressed image
- Some ELF header values are not standard.
  - In particular, *e_machine* must be modifyied for IDA to properly process the file.

Basic Analysis

- IDA will do a good job, but not enough.
- Several functions and string won't be recognized
- Parts of the IOS image were not analyzed correctly.
- Additional analysis is needed.

Enhanced Analysis

- Additional analysis tools written in IDA-Python
- Goal: Detect additional functions & strings.
- Code
  - Explore the whole code segment for unidentified code.
  - All instructions are aligned to 4 bytes.
  - Try to Identify function boundaries.
- Data
  - Look for unidentified strings using a better character set.
  - Try to differentiate references from pure ascii data.
  - Merge some split strings

Results

- On a c2600-i-mz.123-24
- Basic Analysis

28121 Funcs

126379 Strings

- Enhanced Analysis

46296 Funcs

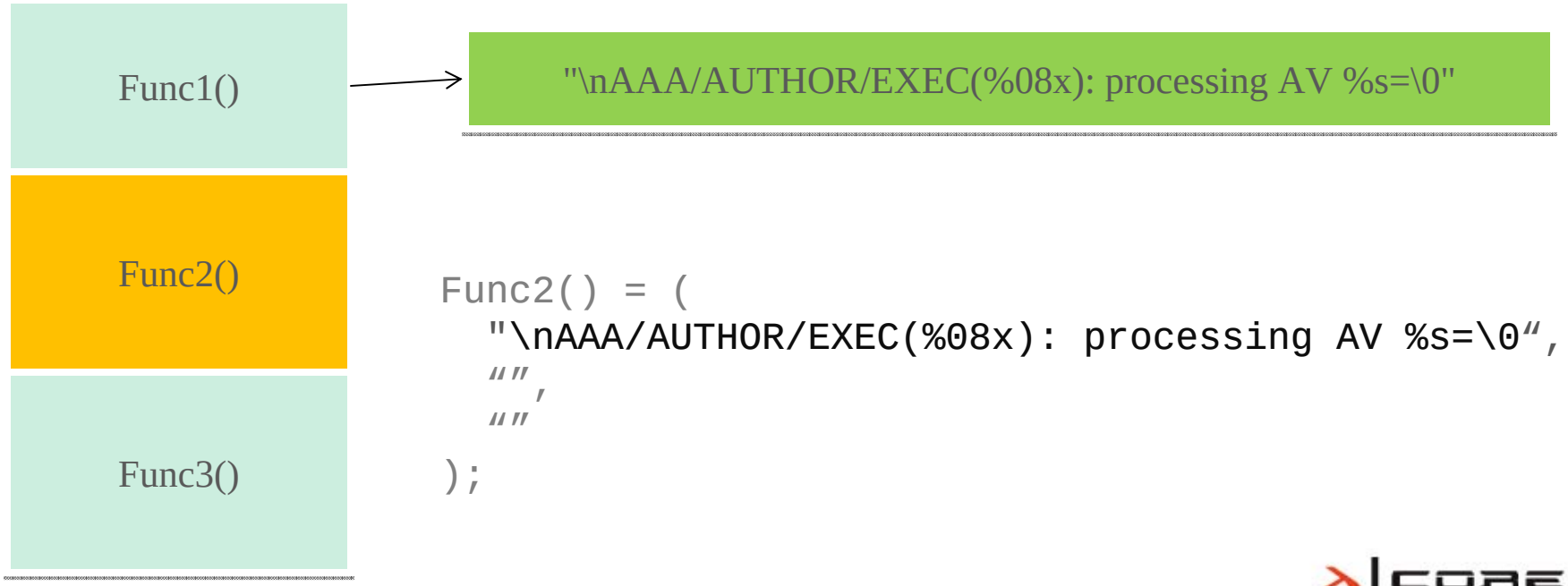
143603 Strings

Targeting low level functions

- We are looking for (offset + prototype).
  - Password checking.
  - File manipulation.
  - Logging information.
  - Packet handling functions.
  - Access lists manipulation function.
- Examples: *socket()*, *recv()*, *open()*, *read()*, *write()*, *etc.*
- Instrumentation code is present, even when usually disabled.
- Lots of descriptive strings are included and could be identified in different versions.
- Call graph, and image layout can also be used

Identifying functions

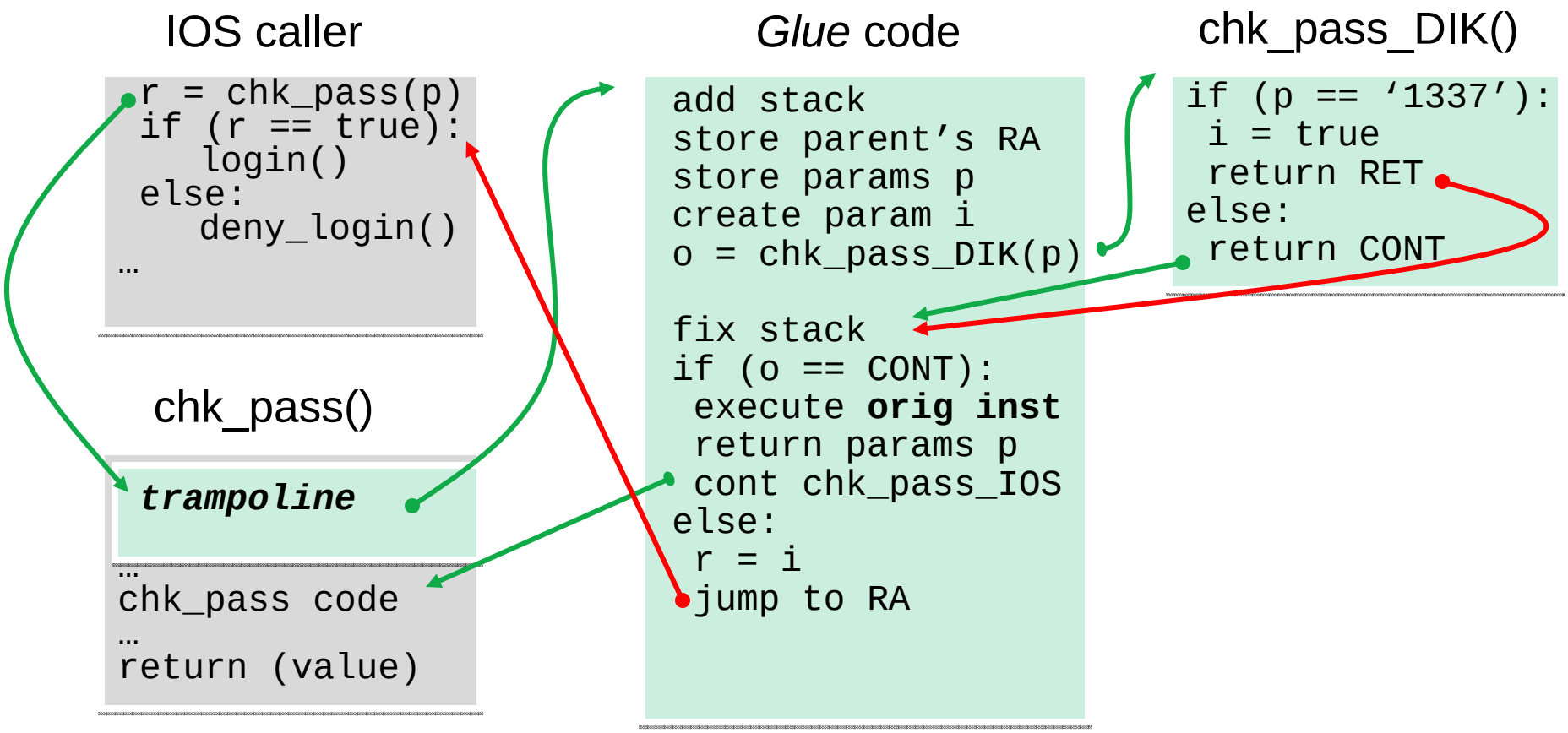
- Functions are described with:
  - String references
  - Function references
  - Neighbor functions



Writing a rootkit function

```
uint chk_pass_DIK(char *input, char *correct, uint val, uint* hook_res)
{
        // my_strcmp is also a rootkit function
        if (my_strcmp(input, pszPassword()) == 0)
        {
                *hook_result = 1; // master password specified
                return OP_RETURN;
        }
        return OP_CONTINUE;
}
```

Linking it all together



Final details

- Almost done:
  - Recompressed the modified image
  - Recalculate Checksums
- Results
  - 80-120 minutes of offline processing
  - All the rootkit “payload” is in c.
  - Code works for PPC & MIPS

Embedded Analysis and Other Scenarios

Upgrade scenario

- Is upgrading to an new IOS version enough to defeat the rootkit?
- Can a compromised router infect a new image on the fly?
- Can a network node infect an image while it is being downloaded.

Exploit payload scenario

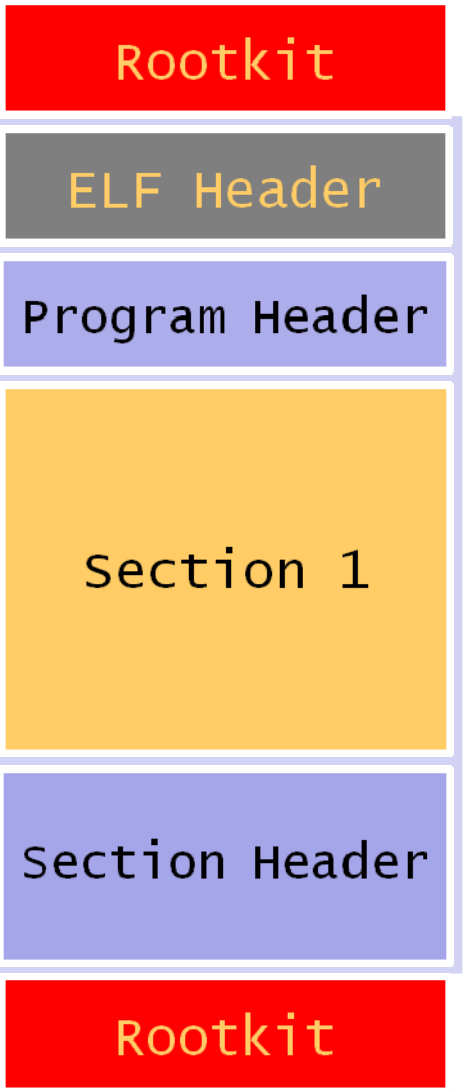
- Exploit reliability could benefit from this type of analysis
- The exploit uses the function recognizer to locate low level IOS functions
- Analysis code size is an important factor!

Additional Constraints

- A direct port of DIK to IOS appears to be difficult
  - Recompression is already too much.
- Embedded Constraints
  - Available memory
  - Processing power
  - Runtime
  - Watchdog times

Very simple Infection

- Can an image be infected by simply adding or appending a small fixed code block and delaying static analysis until needed?
- With only some small changes
- Without the need to recompress



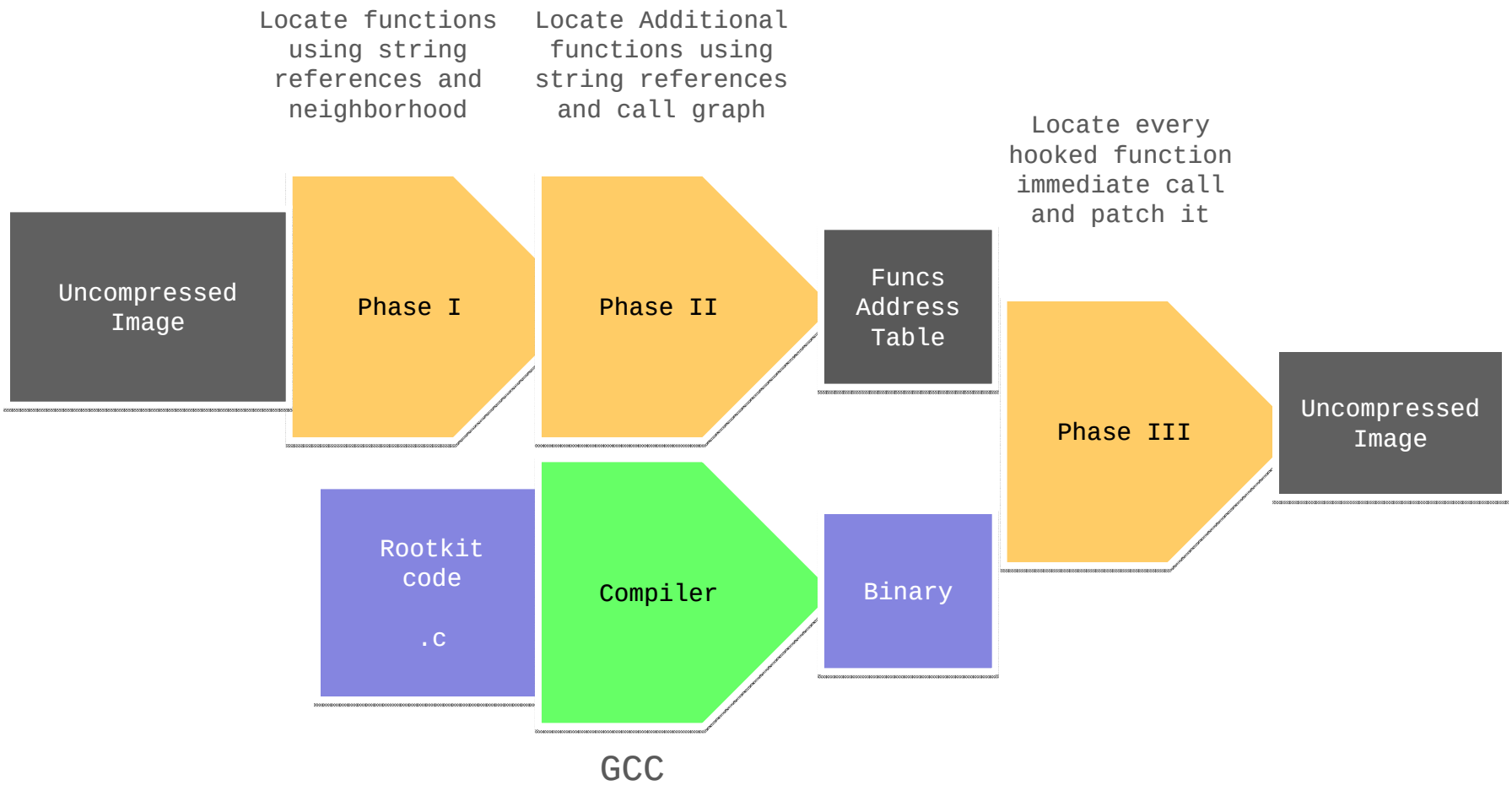
The optimized approach

- We are going to show that
  - A lightweight static analyzer could be implemented
  - The analyzer is fast enough to run unnoticed within bootup time
  - The analyzer is compact enough to be used as exploit payload.
  - Very simple infection is in fact possible
  - A C++ implementation
  - Examples for PowerPC

Lightweight Static Analysis Engine

- Static analysis elements
  - Sweeping over the whole binary image
  - Identify function blocks
  - Identify string references
  - Identify function calls
- Inputs
  - Function description (DIK Style)
  - Rootkit Code: Hook functionality (binary)
- Outputs
  - Identified Functions offsets
  - Image modifications to implement hooks

Architecture



Analysis building blocks / 1

- A string hash function
  - *int code(char *)*
  - Saves space and facilitates comparison
- Sweeping memory is simple:
  - start = *LOADADDRESS*
  - end = *LOADADDRESS+imagelen*
  - 4 byte instructions

```
for(int i = start; i!=end; i+=4) {}
```

Analysis building blocks/2

- Identification macros (PowerPC)

```
#ifdef PPC
#define IS_PRE(x) ( EQ(x,0x94) && EQ((x)+1,0x21)  &&\
        EQ((x)+4,0x7c) && EQ((x)+5,0x08) &&\
        EQ((x)+6,0x02) \        && EQ((x)+7,0xa6)\
 )

#define IS_REF(x) ( \
        EQ((x),0x3c) && EQ((x)+1, 0x60) &&\
        EQ((x)+4,0x38) && EQ((x)+5, 0x63) \
)

#define IS_POST(x) ( \
        EQ(x,0x38) && EQ((x)+1,0x21) && \
        EQ((x)+4,0x4e) && EQ((x)+5,0x80) && \
        EQ((x)+6,0)  &&\       EQ((x)+7,0x20)\
)
#endif
```

```
94 21 FF F8      stwu    %sp, -8(%sp)
7C 08 02 A6      mflr    %r0
90 01 00 0C      stw     %r0, 8+arg_4(%sp)
4B FF 09 41      bl      sub_80632F10
80 01 00 0C      lwz     %r0, 8+arg_4(%sp)
3C 60 81 2B      lis     %r3, addr
38 63 C3 54      addi    %r3, %r3, addr 7
C 08 03 A6       mtlr    %r0
38 21 00 08      addi    %sp, %sp, 8
4E 80 00 20      blr
```

Phase1()

1. The memory sweep loop
2. Using a 3-function look-ahead to detect patterns
3. Strings are identified by its hash code
4. Strings sets of the 3 functions are evaluated on the epilogue and proper offsets stored

```
void phase1(int *func_strs, int *strings, int *func_addr)
{
        for(int i = start; i!=end; i+=4)
        {
                if (IS_PRE(i)) { shift_lookahead(); }
                if (IS_POST(i)) { set func_addr(); }
                if (IS_REF(i)) { lookup(code(i)) }
        }
}
```

Phase2()

- Again, a memory sweep is performed
- Call references to Phase1() identified functions are detected
- String references are used again
- func_addr is filled

```
void phase2(int *func_strs, int *strings, int *func_addr)
{
        for(int i = start; i!=end; i+=4)
        {
                if (IS_PRE(i)) { shift_lookahead(); }
                if (IS_POST(i)) { set func_addr(); }
                if (IS_BL(i)) { looup_func(i); }
                if (IS_REF(i)) { lookup(code(i)) }
        }
}
```

Phase3()

- Instead of using trampoline and glue code as in DIK:
  1. A memory sweep is used to identify every immediate call/branch reference to the target functions
  2. The operand is altered to redirect the calls
- This minimizes the amount of code generated to implement the hook
- The exact same function prototype can be used in the hook function
- Resolved function offsets from previous phases (func_addr) are used to invoke IOS components

Sample function descriptions

```
int strings[]=
{
     code("\n%% Password check with invalid encryption type"),
     code("\nAAA/AUTHOR/EXEC(%08x): processing AV %s=\0"),
     code("\n%s(%08x): tried to change \"service\". Ignore this attribute\0"),
     -1
};

int func_strs[]=
{
     STR_AAA, STR_PASSWORDCHECK, 0,     // password_check
     0, STR_AAA | STR_TRIED, PASSCHECK, // aaa_author_exec
     -1
};
```

Analyzer stats

- Performance:
  - Runtime Complexity is

*O(ImageSize + StrReferences*AvgStrLen + IosFuncs*func_addrs)*

- This notebooks can run ~45.5 infections per second
- Memory footprint is small
  - 4 bytes per string
  - 20 bytes per function
- Code is lightweight
  - PowerPC analysis code is 1180 bytes long
  - MIPS analysis code is 1468 bytes long

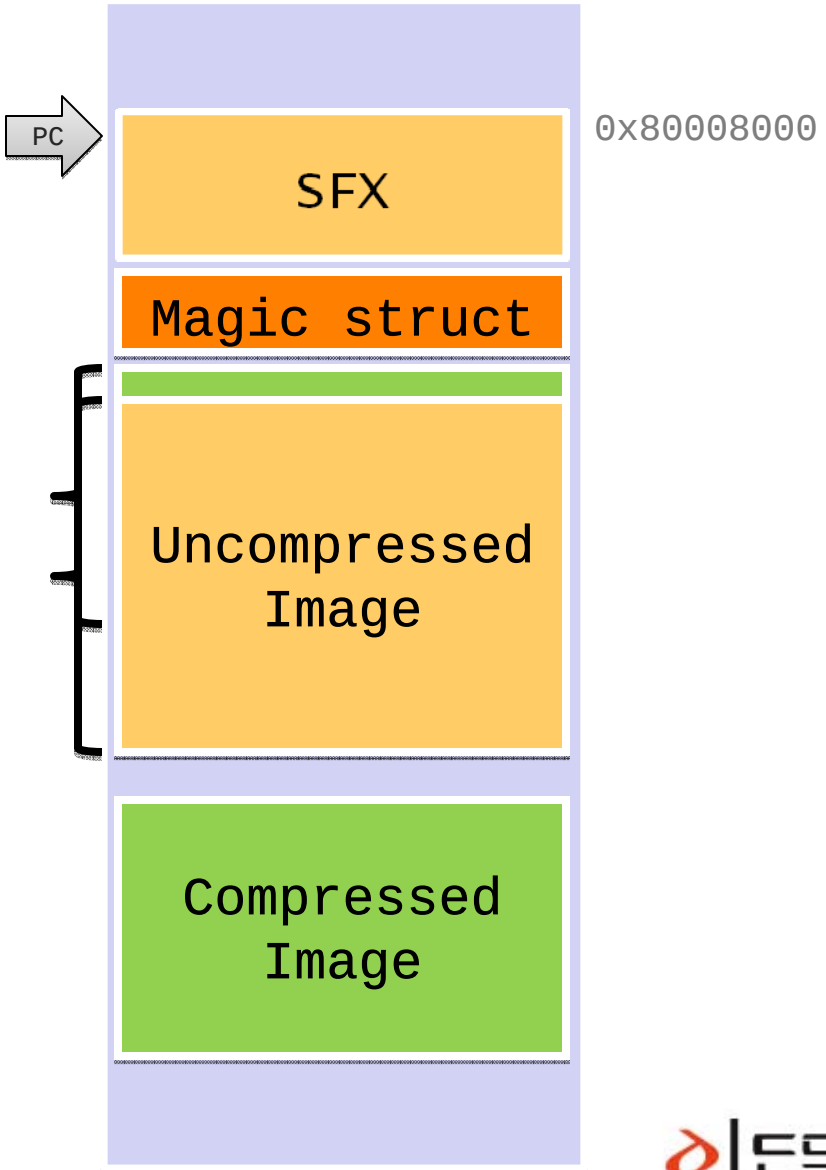
Infecting an image

- We want a generic way to modify a compressed image so
  1. The rootkit payload & analyzer code are included
  2. The analyzer takes control as soon as the image is decompressed
  3. The static analysis magic is performed (and hooks installed)
  4. Execution continues with the decompressed IOS image.

- This is what we called "*Very simple Infection*"

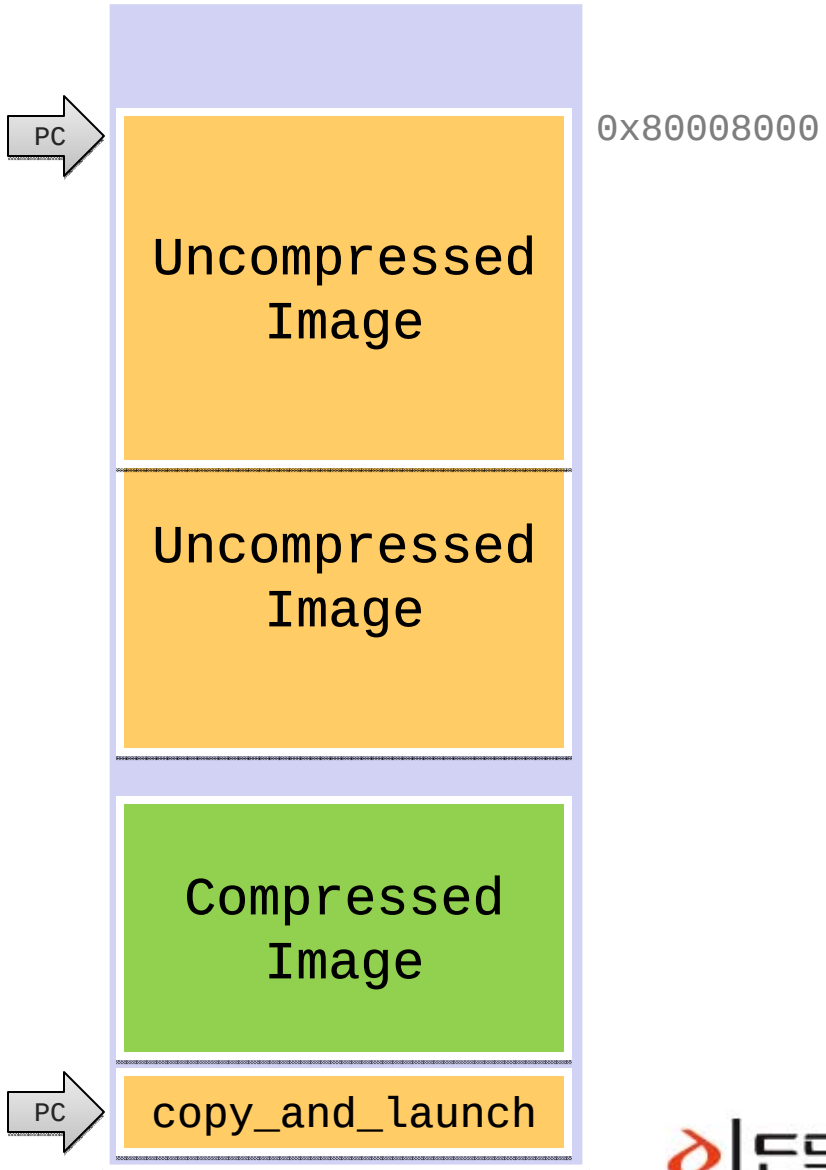
SFX in detail

1. The Compressed elf is loaded into memory and the SFX code is executed
2. Compressed Checksum is verified
3. The compressed image is copied to high memory
4. The image is unpacked
5. Uncompressed Checksum, size and entry point are verified



SFX in detail/2

6. Copy_and_launch() is copied to high memory
7. copy_and_launch is invoked
8. Uncompressed image is moved
9. The uncompressed image is invoked



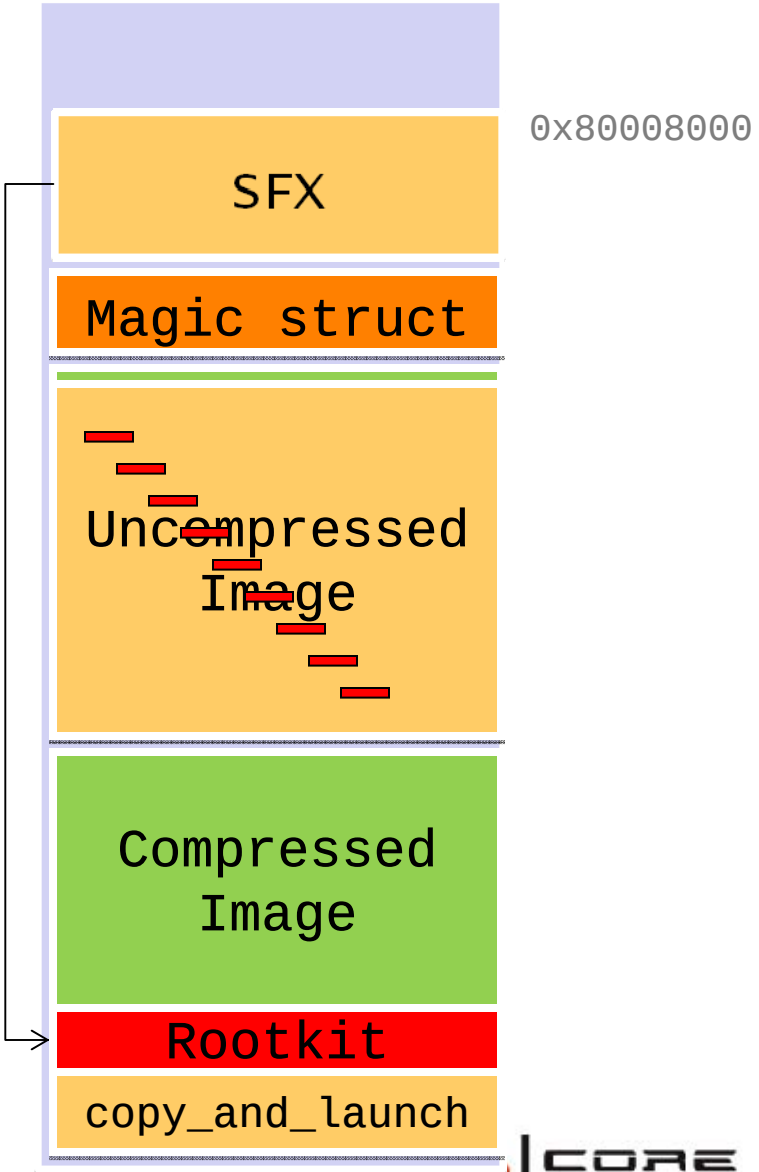
Very simple Infection

File changes

- Analyzer+rootkit code is appended after compressed image
- The compressed image length is modified
- The compressed checksum is modified
- Copy_and_launch call in SFX is modified to invoke the lightweight static analyzer

Runtime

1. The sfx code copies a lot of memory and unpacks the image
2. The analyzer gets control and patches memory
3. Finally, copy_and_launch is invoked



Implications

Therefore…

- This shows that
  - An image infected on the fly,
  - An embedded static analyzer,
  - Static analyzer as an exploit payload,
  - And common virus+rootkit stealth measures

… are all very feasible scenarios,

- sophisticated exploitation and abuse of network devices is not only possible but should be seriously considered in the threat model

So what should we do?

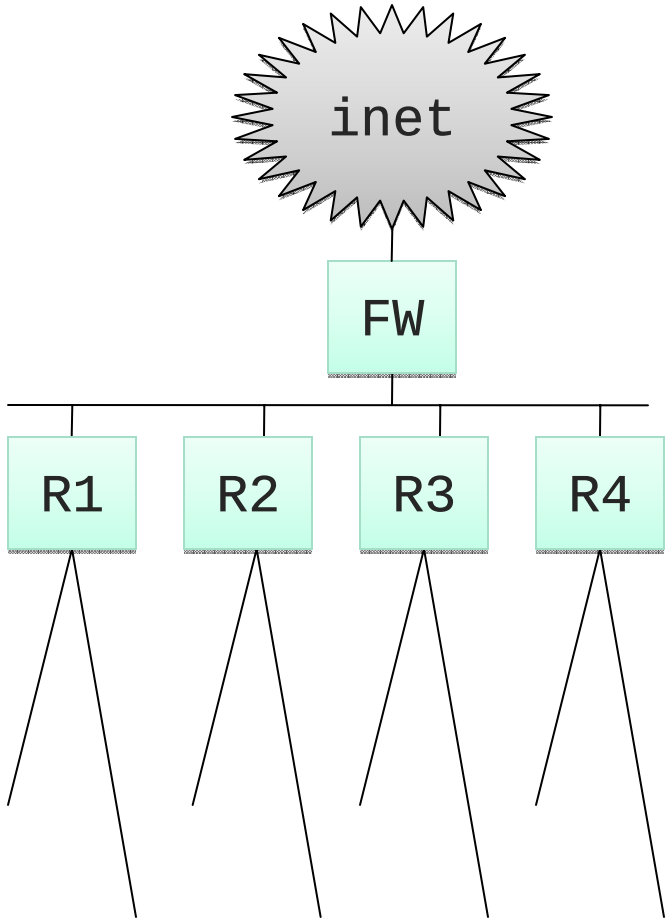
- Verify?
  - How?
- Update?
  - How?
- Check logs?
- Encrypt all my traffic?

Demo

Additional reflections

A complex scenario

- Organization X has several networked locations
- The network is supported on a backbone built on some of the systems targeted on this presentation
- We suspect that some routers have been compromised
- How should we proceed?



Verifying

- Is there a way to verify what's running on a system?
- Is there a way to prove reboot remotely
- Check "Alien vs. Quine", for an interesting approach for a specifically designed embedded system that supports verification of system's memory and running processes

Alien vs. Quine. IEEE Security & Privacy Vol. 5 No. 2
Vanessa Gratzer, Université Paris II Panthéon-Assas
David Naccache, École normale supérieure

Secure logging?

- Why not use some cryptographic tools to make it difficult to hide attack traces.
- Secure logging guarantees that logs generated before the intrusion event cannot be altered without warning the netadmin

```
void OnUpdate(image)
{
    Secret = crypto.hash(secret);
    sign = crypto.sign(image, secret);
    log(sign);
}
```

Crypto could also be...

- Notice that finding a rootkit doesn't mean that you can understand the impact of the incident
- Obfuscation and complex cryptographic protocols could shrink our forensic analysis capabilities
- Imagine that you found a rootkit (from a suspected six month old compromise) with the following code:

```
Void packet_input(packet p)
{
    if (crypto.hash(p) == CRYPTO_CONSTANT)
    {
        decrypt(secretfunction, sizeof(secretfunction), p);
        secretfunction();
    }
};
```

Thank you

ariel.futoransky at coresecurity.com
www.coresecurity.com
www.coresecurity.com/corelabs